# Programación avanzada en Python

## Lección 9: PARALELISMO Y CONCURRENCIA

eip ESCUELA INTERNACIONAL POSGRADOS

# ÍNDICE

# Paralelismo y concurrencia

## PRESENTACIÓN Y OBJETIVOS

En este capitulo veremos dos conceptos muy interesantes y a la misma vez avanzados como son el paralelismo y la concurrencia. Estos dos nos permiten poder controlar como se ejecutan nuestros programas permitiéndonos aumentar la eficiencia en cuanto a tiempo de ejecución se refiere.

### *Objetivos*

- En esta lección aprenderás a:
- Distinguir entre paralelismo y concurrencia
- Que paquetes podemos utilizar para el paralelismo y la concurrencia en Python.
- Como trabajar con estos paquetes.

# 1. PARALELISMO Y CONCURRENCIA

Como breve resumen, podemos decir que ambos conceptos apuntan a ejecutar los programas al mismo tiempo, vamos a ver cada uno de estos conceptos en detalle

## 1.1 Concurrencia

La concurrencia es el proceso de múltiples tareas que se ejecutan simultáneamente y ocurre cuando se produce una situación como la superposición de más de una tarea en algunos recursos.

Se puede entender fácilmente con el diagrama que la concurrencia crea entre las tareas:

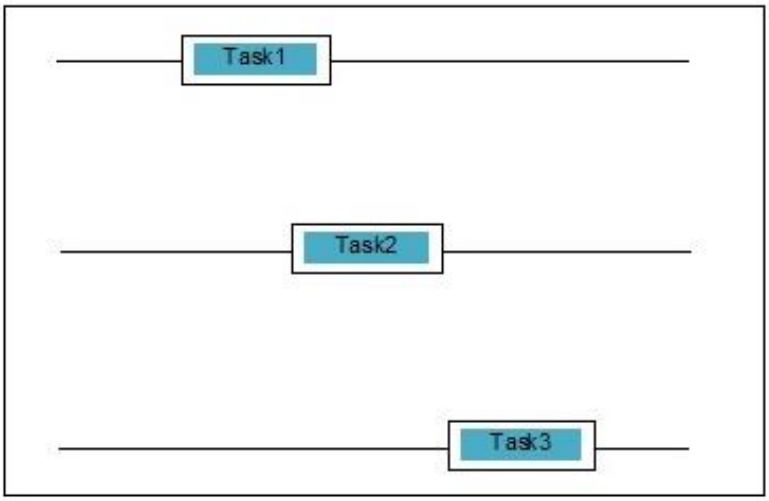


Figura 1.1: Niveles de concurrencia

Hay 3 niveles de concurrencia:

**Concurrencia de bajo nivel:**

Realiza acciones atómicas de forma explícita. No es manejada por Python.

**Concurrencia de nivel medio:**

En lugar de la operación atómica de bajo nivel, utiliza bloqueos explícitamente y es ampliamente utilizada en Python y otros lenguajes. La aplicación puede construirse mediante bloqueos explícitos.

**Concurrencia de alto nivel:**

Aquí no se usan operaciones atómicas ni se usan bloqueos explícitos. Utiliza un tipo de característica y de modelos concurrentes para realizar la concurrencia en Python.

## 1.2 Propiedades

Hay algunas propiedades que se deben conocer:

Correctness:

El programa debe ejecutarse de forma adecuada y correcta, lo que significa que debe de haber una salida en la etapa final.

Seguridad:

Debemos de asegurarnos de que todos los procesos podrán ser ejecutados correctamente.

Live:

El programa debe estar vivo, lo que significa que tiene la capacidad de pasar de un estado a otro.

Actores:

Los procesos y los hilos son actores de los programas concurrentes. Ayudan a hacer una progresión de un estado a otro completando las múltiples tareas simultáneamente.

Recursos:

Para realizar las tareas, los actores procesos e hilos utilizan recursos como la memoria, cpu y otras aplicaciones

Conjunto de reglas:

Hay un conjunto de reglas que preceden al programa de forma concurrente como la asignación de memoria, la asignación de recursos, la modificación del estado, etc. Más adelante hablaremos de todo ello en detalle.

Compartir datos

Es un aspecto importante a tener en cuenta en los programas concurrentes. Algunas variables pueden ser compartidas por algunos programas o módulos. Así que el valor de tales atributos debe ser correcto a lo largo de la tarea. Múltiples hilos y procesos pueden utilizar los datos compartidos o mutables al mismo tiempo.

Por lo tanto, los datos deben ser bloqueados y permitir que sólo un proceso pueda modificarlo a la vez. Se utilizan colas para hacer que los otros procesadores que necesitan los datos esperen en la cola para completar el proceso actual con los datos compartidos.

## 1.3 Paralelismo

El paralelismo o la ejecución de tareas al mismo tiempo ocurre como la división de una tarea en algunas subtareas y su ejecución en paralelo. Veamos un diagrama:

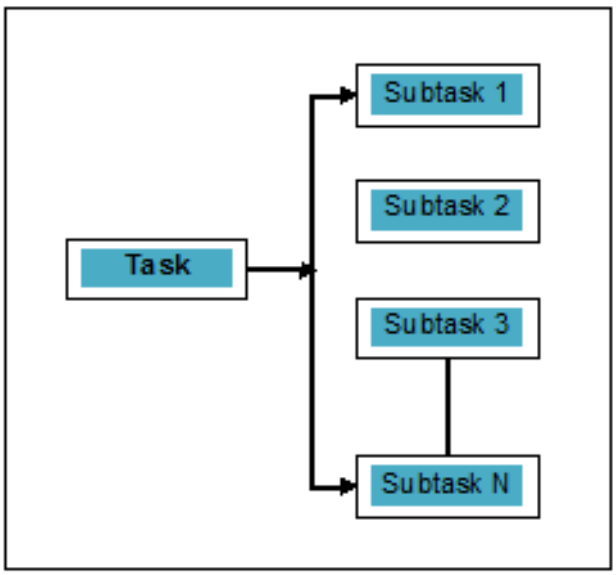


Figura 1.2 Diagrama Paralelismo

La tarea se divide en subtareas y se ejecuta en paralelo para que el programa se ejecute al mismo tiempo.

## 1.4 Threads

Los hilos (threads) son la unidad más pequeña de ejecución, es un solo flujo de control, que se ejecuta dentro del programa.

**Estados de los hilos:**

Hay algunos estados que indican la etapa actual del hilo mientras se ejecuta. Se puede entender fácilmente con el siguiente diagrama:

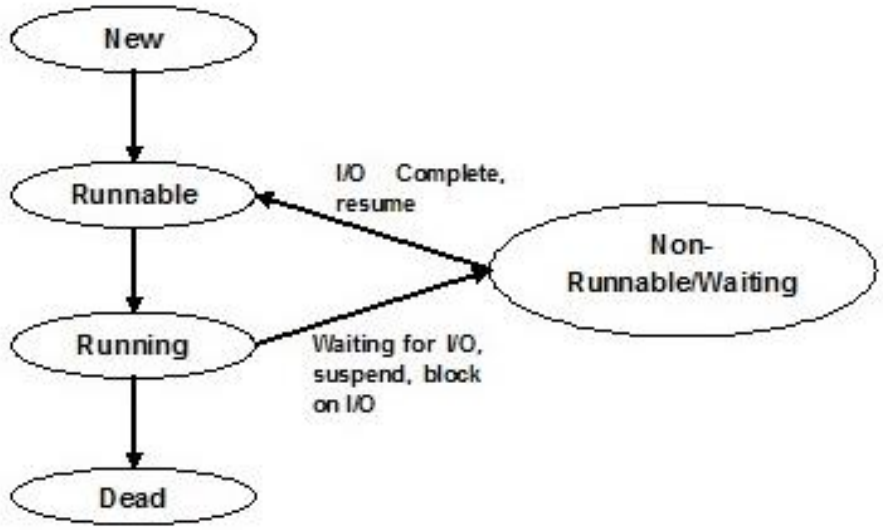


Figura 1.3 Diagrama de la etapa del hilo

- **Nuevo estado**

  Comienzo del ciclo, sólo se crea la instancia para un nuevo hilo y no se asigna ningún recurso.

- **Estado ejecutable**

  La hebra esta lista para ser ejecutada. Está iniciada y esperando a ejecutarse.

- **Estado de ejecución**

  La hebra se está ejecutando, el planificador coge el control y el hilo puede ir a la etapa de espera y al estado muerto.

- **Estado de espera:**

  Si ocurre cualquier interrupción desde cualquier otro recurso o proceso, el hilo actualmente en ejecución pasará al estado de espera para completar la interrupción. Después de completar la interrupción pasa al estado Runnable y cuando el planificador lo selecciona pasa al estado Running.

- **Estado muerto:**

  Cuando un hilo completa su tarea pasa al estado muerto.

# 2. THREAD EN PYTHON

En Python se utilizan dos módulos para implementar los hilos, que ayudan a ejecutar programas al mismo tiempo.

Los paquetes **_thread y threading**

## 2.1 Módulo _thread

Usando el paquete _thread se ejecuta el hilo como una función.

Con la función de start_new_thread() un nuevo hilo es generado. Esta función tiene dos parámetros:

- Función - especificando la función dada como el hilo.
- Args[, kwargs] - argumentos de la función especificados aquí en forma de tupla. Si tiene argumentos de palabra clave, puede dar como un diccionario en kwargs.

Sintaxis:

*_thread.start_new_thread (function.args [, kwargs]*

Veamos un ejemplo:

Importamos el paquete:

```
import _thread
```

A continuación, vamos a definir una función para imprimir los números del 1 al número dado. Se pasan dos parámetros a la función, thread y num. Thread pasa el número de thread y num el número limite que será impreso.

```
def print_num(thread,num):
  for i in range(1,num+1):
    print("Thread", thread,":",i)
```

Ahora vamos a llamar al método start_new_thread con la función y dos argumentos:

```
try:
   _thread.start_new_thread( print_num, (1,3))
except:
   print("Error")
```

Resultado:

```
Thread 1 : 1
Thread 1 : 2
Thread 1 : 3
```

Si se pasa más de un hilo, podemos ver el concepto de concurrencia.

```
try:
   _thread.start_new_thread( print_num, (1,3))
   _thread.start_new_thread( print_num, (2,5))
except:
   print("Error")
```

Como resultado se nos muestran los mensajes desordenados

```
Thread 2 : 1
Thread 2 : 2
Thread 2 : 3
Thread 2 : 4
Thread 2 : 5
Thread 1 : 1
Thread 1 : 2
Thread 1 : 3
```

## 2.2 Módulo threading

Este paquete tiene mayores capacidades que el módulo _thread. Este paquete trata los hilos como un objeto o instancia. Threading tiene las funciones que se encuentran en _thread y también agrega algunas otras funciones como las siguientes:

- activeCount () - número activo de hilos devueltos
- currentThread ()- devuelve el número de hilo.
- enumerate ()- devuelve la lista de todos los hilos que están actualmente activos.

Para implementar hilos utilizando este módulo, debemos de utilizar la clase thread con algunos métodos:

- run () - punto de entrada de un hilo
- start ()- inicia un hilo llamando a run ()
- join([time]) - espera la salida del hilo
- isAlive()- comprueba si el hilo está vivo o sigue ejecutándose
- getName()- devuelve el nombre del hilo
- setName ()- establece el nombre del hilo

Pasos para crear un hilo:

1. Definir un nuevo hilo de clase
2. Añadir argumentos anulando el método init
3. Llamar a run(self, [,args]) para iniciar el hilo
4.

**Estados de los hilos:**

Ya hablamos de los estados de los hilos, estos estados se pueden implementar con las siguientes funciones:

- Importando los paquetes necesarios:

```
import threading
import time
```

- Definiendo la función a llamar en el momento de la creación del hilo

```
def thread_states():
   print("Thread entered in running state")
```

- Usando sleep() para esperar un tiempo

```
time.sleep(2)
```

- Creando un hilo, que toma la función como argumento

```
T1 = threading.Thread(target=thread_states)
```

- Iniciando el hilo con start()

```
T1.start()
Thread entered in running state
```

- Matar el hilo con el método join

```
T1.join()
```

Veamos un ejemplo:

Importando paquetes:

```
import threading
import time
import random
```

Definir una función para llamar con el hilo

```
def Thread_execution(i):
   print("Execution of Thread {} started\n".format(i))
   sleepTime = random.randint(1,4)
   time.sleep(sleepTime)
   print("Execution of Thread {} finished".format(i))
```

A continuación definimos el objeto que nos permitirá crear el hilo con la función y el inicio del hilo:

```
for i in range(4):
   thread = threading.Thread(target=Thread_execution, args=(i,))
   thread.start()
   print("Active Threads:" , threading.enumerate())
```

Resultado:

```
Execution of Thread 0 started
Active Threads:
   [<_MainThread(MainThread, started 6040)>,
      <HistorySavingThread(IPythonHistorySavingThread, started 5968)>,
      <Thread(Thread-3576, started 3932)>]

Execution of Thread 1 started
Active Threads:
   [<_MainThread(MainThread, started 6040)>,
      <HistorySavingThread(IPythonHistorySavingThread, started 5968)>,
      <Thread(Thread-3576, started 3932)>,
      <Thread(Thread-3577, started 3080)>]

Execution of Thread 2 started
Active Threads:
   [<_MainThread(MainThread, started 6040)>,
      <HistorySavingThread(IPythonHistorySavingThread, started 5968)>,
      <Thread(Thread-3576, started 3932)>,
      <Thread(Thread-3577, started 3080)>,
      <Thread(Thread-3578, started 2268)>]
```

```
Execution of Thread 3 started
Active Threads:
   [<_MainThread(MainThread, started 6040)>,
      <HistorySavingThread(IPythonHistorySavingThread, started 5968)>,
      <Thread(Thread-3576, started 3932)>,
      <Thread(Thread-3577, started 3080)>,
      <Thread(Thread-3578, started 2268)>,
      <Thread(Thread-3579, started 4520)>]
Execution of Thread 0 finished
Execution of Thread 1 finished
Execution of Thread 2 finished
Execution of Thread 3 finished
```

Veamos otro ejemplo:

Definimos la clase para el hilo y el método init:

```
class myThread (threading.Thread):
   def __init__(self, threadID, name, counter):
      threading.Thread.__init__(self)
      self.threadID = threadID
      self.name = name
      self.counter = counter
   def run(self):
      print ("Starting " + self.name)
      print_time(self.name, self.counter, 5)
      print ("Exiting " + self.name)
```

El método run () también se define para realizar las tareas especificadas en su interior cuando el hilo se inicia por el método start.

Definimos la función print_time que se utiliza dentro del método run () :

```
def print_time(threadName, delay, counter):
   while counter:
      if exitFlag:
         threadName.exit()
      time.sleep(delay)
      print ("%s: %s" % (threadName, time.ctime(time.time())))
      counter -= 1
```

Ejecutamos nuestro código:

```
thread1.start()
thread2.start()
thread1.join()
thread2.join()
print ("Exiting Main Thread")
```

Resultado:

```
Starting Thread-1
Starting Thread-2
Thread-1: Fri Mar 19 04:31:05 2021
Thread-2: Fri Mar 19 04:31:06 2021
Thread-1: Fri Mar 19 04:31:06 2021
Thread-1: Fri Mar 19 04:31:07 2021
Thread-2: Fri Mar 19 04:31:08 2021
Thread-1: Fri Mar 19 04:31:08 2021
Thread-1: Fri Mar 19 04:31:09 2021
Exiting Thread-1
Thread-2: Fri Mar 19 04:31:10 2021
Thread-2: Fri Mar 19 04:31:12 2021
Thread-2: Fri Mar 19 04:31:14 2021
Exiting Thread-2
Exiting Main Thread
```

**Sincronización de hilos**

Si hay uno o más hilos, entonces todos los hilos deben estar sincronizados correctamente en el caso de compartir datos. De lo contrario, el resultado final será incorrecto o existe la posibilidad de bloquear los procesos. La sincronización es el proceso por el cual dos o más hilos acceden a los recursos al mismo tiempo sin interferir entre sí.

Los principales problemas que surgen con la sincronización son:

- **Bloqueo**:

  El caso de que ningún hilo pueda acceder a ningún recurso.

- **Condición de carrera**:

  El caso de que dos o más hilos puedan acceder a los datos compartidos y traten de cambiar su valor al mismo tiempo.

Para hacerlo sin problemas, se utiliza un concepto llamado sesión crítica.

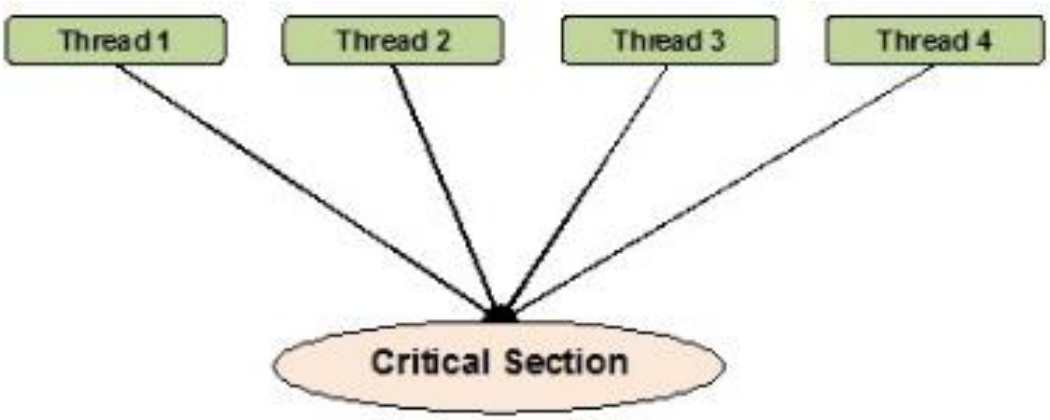


Figura 2.1 Sesión Crítica

Es una parte del programa que contiene la variable compartida o el acceso al recurso. Cuando un hilo entra en la sesión crítica, ésta es bloqueada por el hilo y sólo después de completar su ejecución, cualquier otro hilo puede entrar en ella. Así que en la sesión crítica sólo hay un hilo a la vez.

Veamos algunos ejemplos:

Definimos una variable global:

```
x = 0
```

Definimos una función para incrementar la variable global:

```
def increment_global():

    global x
    x += 1
```

Definimos otra función que llame a la función de incremento:

```
def taskofThread():

    for _ in range(50000):
        increment_global()
```

A continuación, creamos dos hilos con la función taskofThread que utiliza la misma variable, 'x'

```
def main():
   global x
   x = 0

   t1 = threading.Thread(target= taskofThread)
   t2 = threading.Thread(target= taskofThread)

   t1.start()
   t2.start()

   t1.join()
   t2.join()
```

Por último, llamamos a la función principal con un bucle for de 5 iteraciones:

```
if __name__ == "__main__":
   for i in range(5):
      main()
      print("x = {1} after Iteration {0}".format(i,x))
```

Resultado:

```
x = 100000 after Iteration 0
x = 54034 after Iteration 1
x = 80230 after Iteration 2
x = 93602 after Iteration 3
x = 93289 after Iteration 4
```

El valor de x es incorrecto porque es usado por los dos hilos al mismo tiempo.

Hagamos esto con una condición de carrera para obtener un resultado correcto. Para hacer la condición de carrera, los paquetes de hilos utilizan una clase de bloqueo con dos funciones predefinidas:

acquire () :

El método se utiliza para bloquear o no bloquear un bloqueo.

Hay dos argumentos que se utilizan para ello:

- True: por defecto es true y la ejecución del hilo se bloquea hasta que se desbloquee el bloqueo.
- False: la ejecución del hilo se desbloquea hasta que se ponga en valor true.

release () :

El método se utiliza para liberar un bloqueo. Si un bloqueo está bloqueado, la función lo desbloqueará. Si ya está desbloqueado, muestra un error y si hay más de un hilo bloqueado entonces esperará a que se desbloquee.

Vamos a comprobarlo con un ejemplo:

Aquí definimos el taskofThread con el bloqueo y aplicamos las funciones acquire y release para que funcione correctamente.

```
import threading

x = 0

def increment_global():

   global x
   x += 1

def taskofThread(lock):

   for _ in range(50000):
      lock.acquire()
      increment_global()
      lock.release()
```

A continuación, definimos el main con el argumento lock

```
def main():
   global x
   x = 0

   lock = threading.Lock()
   t1 = threading.Thread(target = taskofThread, args = (lock,))
   t2 = threading.Thread(target = taskofThread, args = (lock,))

   t1.start()
   t2.start()

   t1.join()
   t2.join()

if __name__ == "__main__":
   for i in range(5):
      main()
      print("x = {1} after Iteration {0}".format(i,x))
```

El resultado es:

```
x = 100000 after Iteration 0
x = 100000 after Iteration 1
x = 100000 after Iteration 2
x = 100000 after Iteration 3
x = 100000 after Iteration 4
```

## 3. PUNTOS CLAVE

- La **concurrencia** es el proceso de múltiples tareas que se ejecutan simultáneamente y ocurre cuando se produce una situación como la superposición de más de una tarea en algunos recursos.
- El **paralelismo** o la ejecución de tareas al mismo tiempo ocurre como la división de una tarea en algunas subtareas y su ejecución en paralelo.
- Los **hilos (threads)** son la unidad más pequeña de ejecución,